

# A new look at atomic broadcast in the asynchronous crash-recovery model

Sergio Mena      André Schiper

École Polytechnique Fédérale de Lausanne (EPFL)

Distributed Systems Laboratory

CH-1015 Lausanne, Switzerland

Tel.: +41-21-693-5354, Fax.: +41-21-693-6770

{sergio.mena|andre.schiper}@epfl.ch

Technical Report IC/2004/101

## Abstract

Atomic broadcast in particular, and group communication in general, have mainly been specified and implemented in a system model where processes do not recover after a crash. The model is called crash-stop. The drawback of this model is its inability to express algorithms that tolerate the crash of a majority of processes. This has led to extend the crash-stop model to the so-called crash-recovery model, in which processes have access to stable storage, to log their state periodically. This allows them to recover a previous state after a crash.

However, the existing specifications of atomic broadcast in the crash-recovery model are not satisfactory, and the paper explains why. The paper also proposes a new specification of atomic broadcast in the crash-recovery model that addresses these issues. Specifically, our new specification allows to distinguish between a uniform and a non-uniform version of atomic broadcast. The non-uniform version logs less information, and is thus more efficient. The uniform and non-uniform atomic broadcast have been implemented and compared with a published atomic broadcast algorithm. Performance results are presented.

**Keywords:** *Distributed systems, atomic broadcast, crash-recovery model, group communication, fault tolerance*

# 1 Introduction

Atomic broadcast (also called total order broadcast) is an important abstraction in fault tolerant distributed computing. Atomic broadcast ensures that messages broadcast by different processes are delivered by all destination processes in the same order [8]. Many atomic broadcast algorithms have been published in the last twenty years [6]. Almost all of these algorithms have been developed in a model where processes do not have access to stable storage, a model that has been called *crash-stop* (or crash no-recovery). In such a model, a process that crashes loses all its state; upon recovery it cannot be distinguished from a newly starting process. The crash-stop model is attractive from an efficiency point of view: since logging to stable storage is a costly operation, atomic broadcast algorithms that do not log any information are significantly more efficient than atomic broadcast algorithms that access stable storage. However, atomic broadcast algorithms in the crash-stop model also have drawbacks: they tolerate only the crash of a minority of processes. Moreover, there are contexts where access to stable storage is natural, e.g., database systems. It has been shown that replicated database systems can benefit from atomic broadcast [16, 11, 2], but atomic broadcast in the crash-stop model does not suit this context [17].

For this reason, there is a strong motivation to consider atomic broadcast in a model where processes have access to stable storage, a model that has been called *crash-recovery*. In this model, processes have access to stable storage to save part of their state: a process that recovers after a crash can retrieve its latest saved state, and restart computation from there on. Because of the strong link between consensus and atomic broadcast (if one problem is solvable, the other is also solvable), the more basic of these two problems, namely consensus, needs to be addressed first. Among the papers that address crash-recovery consensus [13, 10, 1], we highlight the work by Aguilera *et al* [1]. They define a new failure detector for the crash-recovery model and propose two algorithms for solving consensus in that model. Based on this result, Rodrigues and Raynal address the problem of atomic broadcast in the crash-recovery model [14]. While this paper advances the state-of-the-art, it has a few weaknesses. From our point of view, the main problem in [14] is the specification of atomic broadcast. Classically, atomic broadcast is specified in terms of the two primitives *abcast* (to broadcast a message) and *adeliver* (to deliver a message). No *adeliver* primitive appears in [14], where *adeliver* is a predicate. The value *true* / *false* of the predicate depends on a sequence of messages called *adeliver-sequence*. The application has to poll this sequence for newly delivered messages. This shows a problem in the specification. A more important implication is that the specification in [14] does not

reduce to the classical specification of atomic broadcast in the crash-stop model [8] when crashed processes do not recover.

We point out another limitation of the work in [14]. The work only addresses *uniform* atomic broadcast. *Non-uniform* atomic broadcast in the crash-recovery model is an alternative that may be very interesting from a practical point of view. Non-uniform atomic broadcast can be seen as an intermediate solution, between (1) an atomic broadcast algorithm in the crash-stop model that does not access stable storage at all, and (2) a uniform atomic broadcast algorithm in the crash-recovery model that is expensive due to frequent accesses to stable storage. In contrast to [14], we propose both a *uniform* and a *non-uniform* version of atomic broadcast. The non-uniform atomic broadcast algorithm does not require frequent access to the stable storage. Interestingly, our two specifications reduce to the classical specification of atomic broadcast in the crash-stop model [8] when crashed processes do not recover.

We also explain why atomic broadcast in the crash-recovery model is trickier than in the crash-stop model. Atomic broadcast is most of the time used within an application that has a *state*. The atomic broadcast algorithm also has a *state*. Upon recovery both states must be consistent. However, with no recovery this is not a problem! This becomes a problem in the crash-recovery model. We show how the consistency issue is addressed both in our specification and in our implementation. Finally, we have run experiments that show the gain in performance of the non-uniform version of our atomic broadcast algorithm with respect to the uniform version.

The rest of the paper is organized as follows. Section 2 is devoted to the specification of uniform and non-uniform atomic broadcast in the crash-recovery model. Section 3 discusses the problem of keeping the application state consistent with the state of the atomic broadcast algorithm. Section 4 presents the two algorithms that satisfy our uniform and non-uniform specification of atomic broadcast. Section 5 is devoted to performance evaluation. Finally, Section 6 concludes the paper.

## 2 Specification of Atomic Broadcast in the crash-recovery model

### 2.1 The crash-recovery system model

We consider a system with a finite set of processes  $\Pi = \{p_1, p_2, \dots, p_n\}$ . The system is asynchronous, which means that there is no assumption on message transmission delays or relative speed of processes. The system is *static*, which means that the set  $\Pi$  of processes never changes after system start-up time. During system lifetime, processes can take internal steps or communicate by means of

message exchange.

**Crash and recovery:** Processes can crash and may subsequently recover. We consider system start-up time as an implicit *recover* event. In any process' history, a *recover* event happens always immediately after a *crash* event, except for system start-up time. Moreover, the only event that can happen immediately after a *crash* event (if any) is a *recover* event.

**Up and down:** A process  $q$  is *up* within the segments of its history between a *recover* event and the following *crash* event. If no *crash* event occurs after the last *recover* event in  $q$ 's history, then  $q$  is up forever from its last *recover* event on. In this case, we say  $q$  is *eventually always up*. A process  $q$  is *down* within the segments of its history after a *crash* event until the next *recover* event (if such an event exists).

**Good and bad processes:** A process is *good* if it is *eventually always up*. A process is *bad* if it is not *good*. In other words, a process is *bad* if it (a) eventually crashes and never recovers or (b) crashes and recovers infinitely often.

## 2.2 Definitions

As usual, we define atomic broadcast with an *abcast* and an *adeliver* primitive. We say that process  $q$  *abcasts* message  $m$  if  $q$  executes *abcast*( $m$ ); we say that process  $q$  *adelivers* message  $m$  if  $q$  executes *adeliver*( $m$ ). To these two primitives, we add a third *commit* primitive. Roughly speaking, the commit primitive executed by  $q$  marks the point at which  $q$ 's execution will resume after a crash. When commit is executed by  $q$ , all messages previously *adelivered* at  $q$  will never be *adelivered* again at  $q$ , even if  $q$  crashes and recovers. The commit primitive addresses the fundamental process state problem in the crash-recovery model. The state of each process  $q$  is split into two parts: (1) the application state, and (2) the atomic broadcast protocol state. The distinction between these two states can be ignored when processes do not recover after a crash, but not here. We will come back to this issue later. With the commit primitive, we introduce the following terminology:

1. Process  $q$  *ab-commits* message  $m$  if (1)  $q$  *abcasts*  $m$ , (2)  $q$  executes the primitive *commit*() later on, and (3)  $q$  does not crash in-between.
2. Process  $q$  *del-commits* message  $m$  if (1)  $q$  *adelivers*  $m$ , (2)  $q$  executes the primitive *commit*() later on, and (3)  $q$  does not crash in-between.

We also introduce the notion of *permanent* and *volatile* event. In our model, events are *crash*, *recover*, and the execution of the primitives presented above. If process  $q$  crashes, its volatile events are those that may be lost;  $q$ 's permanent events are those that are not lost even if  $q$  crashes. So if  $q$  never crashes, its whole history is permanent. More formally, the set  $V_q$  of volatile events and the set  $P_q$  of permanent events partition  $q$ 's history, denoted by  $h_q$ . An event  $e \in h_q$  belongs to  $V_q$  if (1) a crash event  $e_c$  occurs after  $e$  in  $h_q$ , and (2) no *commit* event occurs in  $h_q$  between  $e$  and  $e_c$ . An event  $e' \in h_q$  belongs to  $P_q$  if it does not belong to  $V_q$ .

We introduce some additional definitions that will be used in the specification of atomic broadcast:

- *Non-recovery runs*: Let  $R_\Pi$  be the set of all possible runs allowed in the crash-recovery model for process set  $\Pi$ . We define *non-recovery runs* to be the set  $N_\Pi \subset R_\Pi$  of runs that do not contain any commit event or any recover event other than system start-up time.  $N_\Pi$  is the set of all possible runs in the well-known crash-stop model for process set  $\Pi$ .
- *Permanent broadcast*: We say that process  $q$  *permanently abcasts* (or simply  $q$  *p-abcasts*) message  $m$  if (a)  $q$  ab-commits  $m$ , or (b)  $q$  abcasts  $m$  and does not crash later. In other words,  $q$  *permanently abcasts* message  $m$  if event  $abcast(m)$  belongs to set  $P_q$  of  $q$ 's permanent events.
- *Permanent delivery*: Likewise, we say that process  $q$  *permanently adelivers* (or simply  $q$  *p-adelivers*) message  $m$  if (a)  $q$  del-commits  $m$ , or (b)  $q$  adelivers  $m$  and does not crash later. In other words,  $q$  *permanently adelivers* message  $m$  if event  $adeliver(m)$  belongs to set  $P_q$  of  $q$ 's permanent events.
- *Delivery order*: We say that process  $q$  *adelivers message  $m$  before  $m'$*  if (a)  $q$  adelivers  $m$  and later  $m'$  and does not crash in-between, or (b)  $q$  adelivers  $m'$  after having del-committed  $m$ . Notation:  $m \triangleright_q m'$ .

As a result, if  $q$  crashes between the adelivery of  $m$  and  $m'$ , these two messages may not be ordered.

- *Permanent delivery order*: We say that process  $q$  *p-adelivers message  $m$  before  $m'$*  if (1)  $m \triangleright_q m'$  holds, and (2)  $q$  p-adelivers  $m'$ . Notation:  $m \triangleright_p m'$ .<sup>1</sup>
- *Multiple delivery*: We say that process  $q$  *adelivers message  $m$  more than once* if we have  $m \triangleright_q m$ .

As a result, if  $q$  adelivers  $m$  twice but crashes in-between, then  $m$  is not necessarily considered as adelivered more than once.

---

<sup>1</sup>If  $m \triangleright_p m'$  holds, it is easy to see that  $q$  also p-adelivers  $m$ .

## 2.3 Specification of atomic broadcast

We can now formally define atomic broadcast. As in the crash-stop model, we distinguish between *uniform* and *non-uniform* atomic broadcast, a distinction that could not be made in the specification proposed by Rodrigues and Raynal in [14]. A first attempt to introduce this distinction was made in [4] in the context of Reliable Broadcast,<sup>2</sup> but the lack of a primitive like *commit* does not lead to a convincing specification. In our specification, uniform atomic broadcast constrains the behavior of good and bad processes, while non-uniform atomic broadcast does not impose any constraints on (1) bad processes, and (2) volatile events of good processes. In other words, non-uniform atomic broadcast ignores volatile events, and considers only permanent events, i.e., the events that good processes “remember” once they stop crashing.

An important feature of our specification of uniform and non-uniform atomic broadcast is that, in non-recovery runs (see Sect. 2.2), our new specification reduces exactly to the classical definition of uniform and non-uniform atomic broadcast [8]. We define (non-uniform) atomic broadcast by the properties Validity (1), Uniform Integrity<sup>3</sup> (2), Agreement (4), and Total Order (6) defined below. We define uniform atomic broadcast by the properties: Validity (1), Uniform Integrity (2), Uniform Agreement (3), and Uniform Total Order (5).

1. *Validity: If a good process  $q$   $p$ -abcasts  $m$  then  $q$   $p$ -delivers  $m$ .*

There is no uniform Validity property, since it does not make sense to require from a bad process, which can crash and never recover, to deliver  $m$ . So the Validity property is the same for uniform and non-uniform atomic broadcast.

2. *Uniform Integrity: For every message  $m$ , every process  $q$  delivers  $m$  only if some process has abroadcast  $m$ . Moreover,  $m \triangleright_q m$  never holds for any process  $q$ .*

This property allows a process to deliver the same message twice (under certain conditions), unlike Uniform Integrity in the crash-stop model (see the definition of *multiple delivery*, Sect. 2.2). For instance, if process  $q$  delivers message  $m$  and then crashes before del-committing  $m$ , Uniform Integrity allows  $q$  to deliver  $m$  again after recovery.

3. *Uniform Agreement: If a process (good or bad) delivers message  $m$ , then every good process  $p$ -delivers  $m$ .*

---

<sup>2</sup>Reliable Broadcast is weaker than Atomic Broadcast: it does not enforce any order in message delivery.

<sup>3</sup>We do not define a Non-uniform Integrity property, which does not make much sense from a practical point of view.

This property requires that all good processes permanently deliver any message that is delivered by some process. The required permanent delivery of  $m$  ensures that a good process  $q$  *remembers* having delivered  $m$  at the time  $q$  stops crashing.

4. *Agreement: If a good process  $p$  delivers message  $m$ , then every good process  $p$  delivers  $m$ .*

Non-uniform Agreement only puts a constraint on messages  $p$ -delivered by good processes. There is no constraint on a message delivered (but not  $p$ -delivered) by a process that later crashes. Also, there is no constraint on a message  $p$ -delivered by a bad process. In the two cases, no process “remembers”  $m$ .

5. *Uniform Total Order: Let  $p$  and  $q$  be two processes (good or bad). If  $m \triangleright_p m'$  holds and  $q$  delivers  $m'$ , then  $m \triangleright_q m'$  also holds.*

6. *Total Order: Let  $p$  and  $q$  be two good processes. If  $m \triangleright_p m'$  holds and  $q$  delivers  $m'$ , then  $m \triangleright_q m'$  also holds.*

The introduction of the *commit* primitive is fundamental in our specification. It allows us to distinguish between volatile and permanent events. With this distinction it is fairly easy to define uniformity and non-uniformity in the crash-recovery model (and it would be hard to introduce the distinction without the *commit* primitive). In addition, with the *commit* primitive, it is *not* the implementor of atomic broadcast who decides when to make events permanent. This is left to the application, which knows better when volatile events are no more interesting (e.g., because an application checkpoint was taken) and should thus become permanent. Moreover, it is easy to see that, if crashed processes never recover, then our specification of uniform and non-uniform atomic broadcast corresponds exactly to the standard specification of uniform and non-uniform atomic broadcast in the crash-stop model.

The non-uniform specification can be criticised with the argument that a bad process  $p$  (e.g., a process that crashes and recovers infinitely often) can behave arbitrarily, even if it executes *commit* a number of times. However,  $p$  cannot know whether it is good or bad because it may recover in the future and stay up forever. This is similar to the crash-stop model, where a process that crashes in the future is faulty and can thus behave arbitrarily. The practical relevance of non-uniformity is discussed in Section 4.3.

## 2.4 Related work

Atomic broadcast has been specified in the crash-recovery model by Rodrigues and Raynal [14]. They define the primitive  $abcast(m)$  (abroadcast of  $m$ ) and the sequence  $\mu_p = adeliver-sequence()$ . Moreover,  $adeliver(m)$  is a predicate that is true iff  $m \in adeliver-sequence()$  at  $p$ . Atomic broadcast is then specified by the following properties:

- *Validity*: If a process *adelivers* a message  $m$ , then some process has *abroadcast*  $m$ .
- *Integrity*: Let  $\mu_p$  be the delivery sequence at process  $p$ . Any message appears at most once in  $\mu_p$ .
- *Termination*: For any message  $m$ , (1) if the process that issues  $abcast(m)$  returns from  $abroadcast(m)$  and is a good process, or (2) if a process *adelivers* message  $m$ , then all good processes *adeliver*  $m$ .
- *Total order*: Let  $\mu_p = adeliver-sequence()$  at process  $p$ . For any pair of processes  $(p, q)$ , either  $\mu_p$  is a prefix of  $\mu_q$  or viceversa.

This specification has several problems. The main one is the absence of an *adeliver* primitive: how is the *adeliver-sequence* defined? This is the tricky issue that is not addressed. In the group communication literature, specifications usually define the *adeliver* primitive first, and then the *adeliver-sequence* as the sequence of messages *adelivered*. It is the opposite that is done: *adeliver* is defined based on the *adeliver-sequence*, and the *adeliver-sequence* is not defined. Therefore the following statement in the paper is a tautology: “*process p adelivers m if adelivered(m, adeliver-sequence()) is true at p*” Moreover, because of the absence of an *adeliver* primitive, the specification does not reduce to the standard specification of atomic broadcast in the crash-stop model. As a result, all properties derived from the crash-stop model have to be reinvented.

The authors of [14] also mention the following problem with the Validity property. If the call to  $abcast(m)$  returns at a good process  $p$ , this forces all good processes to eventually *adeliver*  $m$ . They argue that this is problematic to ensure if the process crashes shortly after having called  $abcast(m)$ . In contrast, our specification uses the *commit* primitive, which avoids the problem. Our Validity property only forces good processes to eventually *adeliver* message  $m$  if  $p$  *permanently* *abcasts*  $m$  (e.g.,  $p$  *abcasts*  $m$  and then executes *commit*).

Finally, Rodrigues and Raynal also propose an optimized implementation of atomic broadcast. In this implementation, atomic broadcast checkpoints the state of the application from time to time. We claim that, usually, these checkpoints should be initiated by the application (which knows best when



to checkpoint its own state), but this can not be done in the implementation given in [14]. The reason is that the specification lacks a primitive (like *commit*) that the application could use to initiate such a checkpoint.

### 3 Keeping the process state consistent

Atomic broadcast is commonly used to update the state of replicated servers. Consider a replica  $p$ . The state of  $p_i$  needs to be distinguished from the state of the atomic broadcast stack local to  $p$ . We introduce the following notation:  $s_i^{appl}$  denotes the application state of  $p_i$  and  $s_i^{abcast}$  denotes the state of the atomic broadcast stack local to  $p_i$ . We assume here that  $s_i^{appl}$  and  $s_i^{abcast}$  are part of the same OS process denoted by  $p_i$ . The distinction between the  $s_i^{abcast}$  state and the  $s_i^{appl}$  state of  $p_i$  can be completely ignored in the crash-stop model. This is no more the case in the crash-recovery model, where  $p_i$  must recover in a state where  $s_i^{abcast}$  and  $s_i^{appl}$  are consistent. We now address this problem.

#### 3.1 Usage of *commit*

We extend the notation just introduced to denote by  $p_i^{appl}$  the application code of  $p_i$ , and by  $p_i^{abcast}$  the atomic broadcast code of  $p_i$ . In order to recover the state  $s_i^{appl}$  after a crash,  $p_i^{appl}$  checkpoints  $s_i^{appl}$  from time to time. After a crash,  $p_i^{appl}$  recovers in the most recently saved  $s_i^{appl}$  state. From the point of view of  $p_i^{appl}$ , the message delivery sequence should resume exactly where it was at the moment of the checkpoint: the delivery must (1) not include any message logically included in  $s_i^{appl}$ ,<sup>4</sup> (2) but must not miss any message adelivered later in the logical adelivery sequence. For example, consider the logical adelivery sequence  $m_1, m_2, m_3$ . If  $p_i^{appl}$  has checkpointed its state after the adelivery of  $m_1$  and crashed after the handling of  $m_2$ , then the delivery after recovery should restart with  $m_2$ . The *commit* primitive naturally fits this requirement: process  $p_i^{appl}$  checkpoints  $s_i^{appl}$  and then immediately executes *commit*. Condition (1) above is guaranteed by the (Uniform) Integrity property (which ensures that no del-committed message will be adelivered again); condition (2) is ensured by the Agreement property.

This solution works as long as the checkpoint and the commit operations are executed atomically, that is, a process can never crash between  $t_1$  and  $t_2$  in Figure 1. Moreover, all events (depicted as circles in Fig. 1) are assumed to be atomic so far. We now explain how these two assumptions can be relaxed, while keeping  $s_i^{appl}$  and  $s_i^{abcast}$  consistent upon recovery.

---

<sup>4</sup>A message  $m$  is logically included in the checkpointed state if it led to the update of  $s_i^{appl}$ .

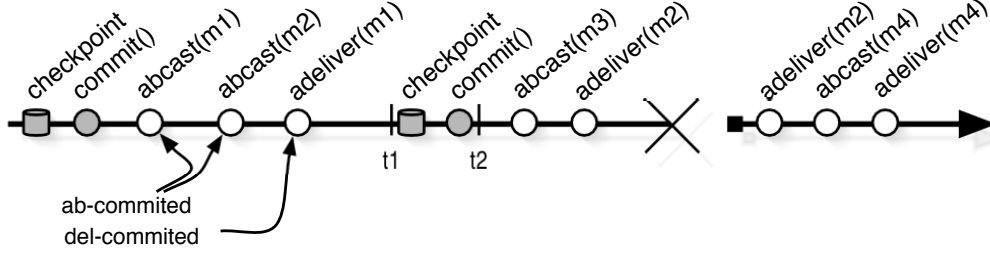


Figure 1: Example execution. After  $p_i^{appl}$  checkpoints its state, it calls *commit*.

### 3.2 Addressing the atomicity problem

For a process  $p_i$ , we have introduced the distinction between  $p_i^{appl}$  and  $p_i^{abcast}$ . The interaction between  $p_i^{appl}$  and  $p_i^{abcast}$  is naturally expressed by means of function calls (e.g., *abcast* function, *commit* function). Function calls are synchronous: the caller blocks while the call is being executed. This yields a useful property: the caller is sure that the callee has completely processed the call when it returns. Consider now that  $p_i$  crashes during the function call (e.g., during *abcast* or *commit*). When  $p_i$  recovers, it does not know whether the function was successfully executed or not.

To address this problem, we model the communication between  $p_i^{appl}$  and  $p_i^{abcast}$  in terms of *messages*. When  $p_i^{appl}$  invokes the primitive  $F(\text{PARAMETERS})$  on the atomic broadcast interface, we say it sends the (local) message  $\langle F, \text{PARAMETERS} \rangle$  to  $p_i^{abcast}$  (see Fig. 2). Likewise, when  $p_i^{abcast}$  invokes  $F'(\text{PARAMETERS}')$  on the application interface, we say it sends the (local) message  $\langle F', \text{PARAMETERS}' \rangle$  to  $p_i^{appl}$  [17].

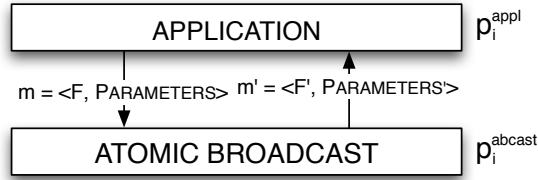


Figure 2: Function calls and callbacks can be modeled as messages.

When modelling intra-process communication using the message-passing model, a single process  $p_i$  becomes a *distributed system* with two processes  $p_i^{appl}$  and  $p_i^{abcast}$ . If we represent Figure 1 using this message-passing model, it becomes Figure 3. The atomicity problem now becomes the problem

to recover  $p_i^{appl}$  and  $p_i^{abcast}$  in a consistent global state.

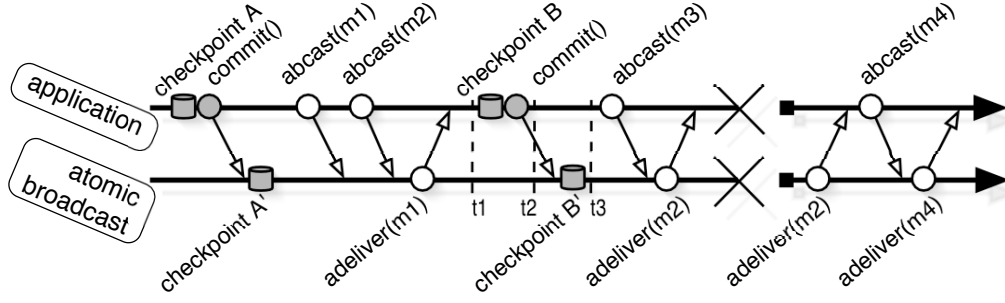


Figure 3: Expressing Figure 1 using message passing communication.

This modelling allows us now to apply results from the checkpointing literature [7]. A message  $m$  becomes *orphan* when its sender is rolled-back to a state before the sending of  $m$  ( $m$  is *unsent*), but the state of its receiver still reflects the reception of  $m$ . In this case, the receiver is said to be an orphan process. Orphan processes cannot be tolerated: the orphan process needs to be rolled-back (even if it did not crash). A message  $m$  is *in-transit* when its receiver is rolled-back to a state before the reception of  $m$  ( $m$  is *unreceived*), while the sender is in a state in which  $m$  was sent. *In-transit* messages are tolerated under the condition that the rollback-recovery protocol is built on top of lossy channels [7]. In our model communication is reliable, so we cannot recover in a state with in-transit messages. Therefore, in-transit messages cannot be tolerated, either.

We now discuss how to recover  $p_i^{appl}$  and  $p_i^{abcast}$  in a global state with no orphan and no in-transit messages. We explain the solution on Figure 3. Consider the second *checkpoint-commit* pair. We only have to distinguish three cases: (1) crash at  $t1$ , i.e., before the checkpoints  $B$  and  $B'$ , (2) crash at  $t2$ , i.e., after checkpoint  $B$  but before checkpoint  $B'$ , and (3) crash at  $t3$ , after the checkpoints  $B$  and  $B'$ .

Cases (1) and (3) are analog: no in-transit and orphan messages in the global state  $(A, A')$  and in the global state  $(B, B')$ . Thus, we only discuss case (3), which is depicted in Figure 3. Since there is no in-transit message in the global state  $(B, B')$ ,  $p_i^{appl}$  is rolled back to the checkpoint  $B$  and  $p_i^{abcast}$  is rolled back to the checkpoint  $B'$ .

In case (2), the global state  $(B, A')$  contains at least one in-transit message (the commit), and so  $p_i$  cannot be rolled-back to this state. So  $p_i^{appl}$  is forced to rollback to checkpoint  $A$  and  $p_i^{abcast}$  is rolled-back to checkpoint  $A'$ .

So the only problem is to know whether case (2) or (3) occurs. This can easily be done by

counting the number of  $s_i^{appl}$  checkpoints and the number of  $s_i^{abcast}$  checkpoints. If the two numbers are equal we are in case (3). Otherwise, we are in case (2). Algorithm 1 shows the corresponding pseudo-code. At the atomic broadcast level, i.e.,  $p_i^{abcast}$ , the variable  $nb\_commits$  counts the number of commits executed so far. Its value is logged with the data that the commit procedure logs, so it really reflects the number of commits executed despite crashes. At the application level, i.e.,  $p_i^{appl}$ , the array  $st$  represents the sequence of checkpoints of  $s_i^{appl}$ .<sup>5</sup> The variable  $nb\_checks$  keeps track of the number of checkpoints done so far. It is important that no message is delivered during the checkpointing phase (lines 4 through 8). Upon recovery,  $st$  is retrieved and the value  $nb\_checks$  is computed (line 10). Then,  $p_i^{appl}$  queries  $p_i^{abcast}$  to find out whether (a) it can resume execution from its very last checkpoint, or (b) it has to roll back to the previous checkpoint. Note that actually  $p_i^{appl}$  only keeps the two most recent checkpoints.

---

**Algorithm 1** Keeping local consistency between atomic broadcast and the application .

---

1: <b>At application level</b>	14: <b>At atomic broadcast level</b>
2:   Initialisation:	15:   Initialisation:
3: $nb\_checks \leftarrow 0; \forall i \in \mathbb{N} : st[i] \leftarrow \perp$	16: $\dots; nb\_commits \leftarrow 0; \dots$
4: <b>to</b> checkpoint( $state$ )	17: <b>procedure</b> get_nb_commits()
5: $nb\_checks \leftarrow nb\_checks + 1$	18: <b>return</b> ( $nb\_commits$ )
6: $st[nb\_checks] \leftarrow state$	
7: $st[nb\_checks - 2] \leftarrow \perp$	19: <b>upon</b> commit() <b>do</b>
8:   log( $st$ ); abcast.commit()	20: $\dots; nb\_commits \leftarrow nb\_commits + 1$
	21:    log $nb\_commits$ together with other data; $\dots$
9: <b>upon</b> recovery <b>do</b>	
10:   retrieve( $st$ ); $nb\_checks \leftarrow \max\{i : st[i] \neq \perp\}$	22: <b>upon</b> recovery <b>do</b>
11: <b>if</b> abcast.get_nb_commits() < $nb\_checks$ <b>then</b>	23: $\dots$ ; retrieve( $nb\_commits$ ); $\dots$
12: $st[nb\_checks] \leftarrow \perp$ ; log( $st$ )	
13: $nb\_checks \leftarrow nb\_checks - 1$	

---

## 4 Solving uniform and non-uniform atomic broadcast

There are several alternatives to solving atomic broadcast in the crash-recovery model, in the same way as there are various algorithms that have been proposed to solve it in the crash-stop model [6]. In this section, we have chosen to illustrate how to implement the new atomic broadcast specifications of Section 2 by reduction to a sequence of consensus. This technique is well accepted in the crash-stop model, which justifies our choice. We first present an algorithm that implements uniform atomic broadcast in the crash-recovery model. Then, we discuss how to convert this algorithm into a more efficient one that satisfies the weaker non-uniform atomic broadcast specification. Both algorithms

---

<sup>5</sup>Representing this as an infinite sequence simplifies the presentation.

solve the problem by reduction to consensus.

## 4.1 Building blocks

The algorithms we present below rely on the following building blocks.

**Logging.** During normal execution, processes use non-persistent memory to keep their state. They access stable storage from time to time to save data from non-persistent memory. When a process crashes and later recovers, only the data saved to stable storage is available for retrieving. A process uses function  $\text{log}(X)$  to log the content of variable  $X$  to stable storage, and the function  $\text{retrieve}(X)$  to retrieve (upon recovery) the previously logged value of  $X$ . These two functions are very costly and should be used as sparsely as possible.

**Fair-lossy channels.** Processes communicate using channels. Because of the crash-recovery model, we cannot assume reliable channels. Indeed, consider processes  $p$  and  $q$ : if  $p$  sends a message  $m$  to  $q$  while  $q$  is down, the channel cannot deliver  $m$  to  $q$ . So we assume fair-lossy channels and the two communication primitives:  $\text{send}(\text{message})$  to *destination* and  $\text{receive}(\text{message})$  from *source*. They ensure the following property: if  $p$  sends an infinite number of messages to  $q$  and  $q$  is good, then  $q$  receives an infinite number of messages from  $p$ . Fair-lossy channels can be implemented without access to stable storage.

**Consensus.** The algorithms below solve atomic broadcast by reduction to consensus, i.e., we need a building block that solves consensus. In consensus, each process proposes a value, and (1) all good processes decide a value, (2) this value is the same for all processes that decide,<sup>6</sup> and (3) it is the initial value proposed by some process. Section 4.4 discusses how to solve consensus.

## 4.2 Uniform atomic broadcast

**Overview.** Algorithm 2 implements the uniform variant of our atomic broadcast specification. The algorithm reduces atomic broadcast to a sequence of consensus as in [5] for the crash-stop model. It is also influenced by the algorithms in [14] (which are actually derived from [5]). The algorithm has two tasks: the *sequencer* task and the *gossip* task. The *sequencer* task executes a sequence of consensus to decide on the delivery order of messages, while logging every value proposed to stable storage. The

---

<sup>6</sup>Actually, this defines uniform consensus. In this paper, consensus always stands for *uniform* consensus. Note that the specification of non-uniform consensus in the crash-recovery model [1] is not well-adapted for this work.

*gossip* task is responsible for disseminating new messages among all processes. This is necessary to ensure eventual message reception with fair-lossy channels. When *commit* is executed, the algorithm also logs the part of its state that is necessary in the case of a crash followed by a recovery. Upon recovery, the algorithm “replays” (see lines 11 to 16) all messages *adelivered* beyond the most recent *commit* executed before the crash. This is needed in order to satisfy the specification of uniform atomic broadcast.

**Innovations.** Since the basic idea of the atomic broadcast algorithm is inspired by [5] and [14], we found it inappropriate to explain the details. More interesting is to focus on the differences. Thus, we now explain the main differences between Algorithm 2 and the algorithm presented in [14] (and we also point out a bug in this algorithm, see below). These differences are marked with grey background (e.g., line 6, line 10, etc.). The rectangles surrounding part of the code should be ignored for the moment (e.g., lines 11 to 16): they are discussed in Section 4.3.

---

**Algorithm 2** Solving uniform atomic broadcast. The non-uniform algorithm is obtained by removing the code inside the white boxes.

---

<pre> 1: <b>For every process</b> <math>p</math> 2:   Initialisation: 3:   <math>\forall i \in \mathbb{N} : Proposed[i] \leftarrow \perp</math> 4:   <math>Unord \leftarrow \emptyset ; A\_deliv \leftarrow \emptyset</math> 5:   <math>k \leftarrow 0 ; gossip\_k \leftarrow 0</math> 6:   <math>nb\_commits \leftarrow 0</math>  7:   <b>procedure</b> process_decision(<math>decision</math>) 8:     <math>result \leftarrow decision \setminus A\_deliv</math> 9:     <math>A\_deliv \leftarrow A\_deliv \cup result</math> 10:    <math>adeliver(result) ; k \leftarrow k + 1</math>  11:   <b>procedure</b> replay() 12:     <b>while</b> <math>Proposed[k] \neq \perp</math> <b>do</b> 13:       <math>Unord \leftarrow Unord \cup Proposed[k]</math> 14:       propose(<math>k, Proposed[k]</math>) 15:       <b>wait until</b> decide(<math>k, decision</math>) 16:       process_decision(<math>decision</math>)  17:   <b>upon</b> initialization <b>or</b> recovery <b>do</b> 18:     retrieve(<math>k, A\_deliv, Unord, nb\_commits</math>) 19:     fork_task(gossip) 20:     retrieve(<math>Proposed</math>); replay() 21:     fork_task(sequencer) </pre>	<pre> 22: <b>upon</b> A-broadcast(<math>m</math>) <b>do</b> 23:   <math>Unord \leftarrow Unord \cup \{m\}</math>  24: <b>upon</b> commit() <b>do</b> 25:   <math>nb\_commits \leftarrow nb\_commits + 1</math> 26:   log(<math>k, A\_deliv, Unord, nb\_commits</math>)  27: <b>upon</b> receive(<math>k', Unord'</math>) from <math>q</math> <b>do</b> 28:   <math>Unord \leftarrow Unord \cup Unord' \setminus A\_deliv</math> 29:   <math>gossip\_k \leftarrow \max(gossip\_k, k')</math>  30: <b>task</b> Gossip 31:   <b>repeat forever</b> 32:     send(<math>k, Unord</math>) to all  33: <b>task</b> Sequencer 34:   <b>repeat forever</b> 35:     <b>wait until</b> <math>Unord \neq \emptyset</math> <b>or</b> <math>gossip\_k &gt; k</math> 36:     <math>Proposed[k] \leftarrow Unord</math> 37:     log(<math>Proposed[k]</math>) 38:     propose(<math>k, Proposed[k]</math>) 39:     <b>wait until</b> decide(<math>k, decision</math>) 40:     process_decision(<math>decision</math>) 41:     <math>Unord \leftarrow Unord \setminus A\_deliv</math> </pre>
--	---

---

The only new variable is *nb\_commits* (line 6), which counts the number of *commits* locally performed since system start-up time (see Section 3). This variable is accessed in lines 18, 25, and 26.

Lines 24 through 26 are executed upon *commit*. Commit saves to stable storage all data necessary

to restore its state upon recovery. These data are (1) the number of the current instance of consensus (or the next one, if there is no consensus running at the local process), (2) the variable  $A_{deliv}$  containing messages already delivered, (3) the set  $Unord$  of messages received but not yet delivered<sup>7</sup>, and (4) the variable  $nb\_commits$  defined above. The rest of the state is either not needed: variable  $gossip_k$ , or logged elsewhere: the array  $Proposed$  (values proposed to consensus).

Note that, unlike [14], our algorithm *does* include the primitive *adeliiver* (see Section 2). *Adeliiver* occurs every time a message is added to set  $A_{deliv}$ , i.e., in line10.

Upon recovery, procedure *Replay* proposes again the initial values that were proposed before the crash. It does so in line 14. This line is necessary because we assume the consensus specification in [1], and thus, for each consensus  $k$  that decided before the crash, we need to propose the same value upon recovery to have the guarantee that consensus  $k$  decides again after the recovery. Line 14 would not be necessary if consensus was also specified with the commit primitive.

Finally, line 19 differs from [14]: it is incorrect in the optimized algorithm in [14]<sup>8</sup>.

The correctness argument of Algorithm 2 is similar to [14].

### 4.3 Non-uniform atomic broadcast

Informally, the difference between the uniform and non-uniform atomic broadcast algorithms is that the non-uniform algorithm only needs to write to stable storage upon execution of *commit*. The uniform algorithm presented in the previous section needs to log every value proposed to consensus (Algorithm 2, line 37). The reason is that volatile events have to be replayed upon recovery, exactly as they occurred before the crash. The uniform algorithm thus accesses the stable storage every time an instance of consensus is started. In contrast, the non-uniform algorithm can *forget* volatile events at any process, while still fulfilling its specification. Thus, if a process crashes and recovers, it only needs to remember its state at the time of the last commit. Applications that can afford losing uncommitted parts of the execution can typically benefit from non-uniform atomic broadcast. Note that the total order and agreement properties *do hold* at good processes even if processes forget volatile events when crashing, so the application state does not become inconsistent at those good processes. The non-uniform algorithm is easily derived from Algorithm 2 by removing the code in the white boxes (e.g., lines 11 to 16, line 20, etc.).

Access to stable storage is extremely expensive and should be used as sparsely as possible. Thus,

---

<sup>7</sup>This differs from [14], where this information is logged every time a new message is abcast, which is less efficient.

<sup>8</sup>If line 19 is placed after calling *replay*, the latter may block forever.

if the application does not execute *commit* frequently, the performance of the non-uniform algorithm is highly improved compared to the uniform algorithm. Furthermore, if the underlying consensus algorithm does not access the stable storage too frequently, the performance of non-uniform atomic broadcast can even be close to a crash-stop atomic broadcast algorithm. We discuss this issue in the next section.

#### 4.4 Which consensus algorithm should be used?

Both atomic broadcast algorithms presented in the previous section require an algorithm solving consensus in the crash-recovery model. Aguilera *et al* propose two such algorithms [1]: one of them accesses stable storage, whereas the other does not.

**Consensus with access to stable storage.** The consensus algorithm with access to stable storage is well suited for uniform atomic broadcast. It solves consensus as long as a majority of processes are good, but accesses the stable storage very often: (1) every time the state changes locally, and (2) when the process decides. The stable storage is thus accessed at least twice per consensus. This does not impact performance of uniform atomic broadcast as much as one could think, since the uniform atomic broadcast itself logs its proposed value at the beginning of every consensus.

However, using this algorithm with our non-uniform atomic broadcast is overkill: it reintroduces frequent access to stable storage that we managed to suppress with our non-uniform algorithm, i.e., performance of the non-uniform atomic broadcast becomes poor: the performance of non-uniform atomic broadcast algorithm is almost the same as the performance of the uniform atomic broadcast algorithm.

**Consensus without access to stable storage.** Aguilera *et al* show that consensus can also be solved in the crash-recovery model without accessing stable storage. This consensus algorithm suits our non-uniform atomic broadcast in the sense that it does not reduce performance, as it does not access stable storage. With this solution, we achieve our goal of avoiding access to stable storage as long as *commit* is not executed. However, the algorithm requires the number of *always-up* processes to be larger than the number of bad processes [1]. This is not a big constraint from a practical point of view. Indeed, by having *commit* log part of the state of the consensus algorithm, the *always-up* processes are only required to stay up between two consecutive *commits*.



## 5 Performance evaluation

We have implemented different atomic broadcast algorithms to compare their performance for various group sizes:  $n = 3$  and  $n = 7$ .<sup>9</sup> The algorithms implemented are: (a) the optimized uniform atomic broadcast algorithm proposed by Raynal and Rodrigues [14], (b) the uniform atomic broadcast algorithm of Section 4, (c) the non-uniform atomic broadcast algorithm of Section 4, and (d) a well-known uniform atomic broadcast algorithm in the crash-stop model [5]. All these algorithms reduce atomic broadcast to a sequence of consensus: algorithms (a) and (b) use a crash-recovery consensus algorithm that accesses stable storage (see Section 4.4), algorithm (c) uses a crash-recovery consensus that does not access stable storage (see Section 4.4), algorithm (d) uses a crash-stop consensus algorithm [5]. All algorithms were implemented in Java and follow the conventions of our Fortika framework [12]. These conventions allow protocol composition with different composition frameworks. We used the Cactus [3, 9] framework for these experiments. Algorithms (a), (b) and (c) use the same libraries for stable storage and for fair-lossy channels. Algorithm (d) uses TCP-based reliable channels.

The hardware used for the measurements was (1) a 100 Base-TX Ethernet, with no third-party traffic, (2) seven PCs running Red Hat Linux 7.2 (kernel version 2.4.18-19). The PCs have a Pentium III 766Mhz processor, 128 MB of RAM, and a 40 GB (Maxtor 6L040J2) hard disk drive . The Java Virtual Machine was Sun's JDK 1.4.0.

In our experiments, the first process in the group<sup>10</sup> steadily abcasts 128-byte-long messages.<sup>11</sup> The offered load was constant at 100 messages per second (i.e., the benchmark *tries to* abcast 100 messages per second, but protocol flow control will block it from time to time). The actual throughput was less than that, since the sending thread is blocked when there are too many messages in the local set *Unord* (Algorithm 2, line 23). Besides, all processes execute *commit* every  $t$  seconds, where  $t$  ranged from 100 milliseconds to 5 seconds. In each experiment, we measured the average *early latency* of messages after the execution became stationary. The *early latency* for message  $m$  is the time elapsed between the abcast of  $m$  and first adelivery of  $m$  [15]. We also measured the average throughput, defined as the number of messages adelivered per second. Note that the experiments where performed with no crashes and no false crash suspicions. The main goal of these experiments was to see how the performance is affected as the frequency of commits increases.<sup>12</sup>

---

<sup>9</sup>We did the same tests for  $n = 5$  and the results are in-between.

<sup>10</sup>The process with the smallest id.

<sup>11</sup>We have also done the same experiments with multiple senders; yielding similar results.

<sup>12</sup>In [14], a checkpoint is taken instead of a commit, which is the same in terms of the implementation.

The early latency results are shown in Figure 4, with the 95% confidence interval. As expected, Rodrigues-Raynal and our uniform algorithm perform similarly since both of them use the stable storage for every consensus. An important observation is that the non-uniform algorithm performs much better than the two uniform algorithms when commits are not frequent, since it only accesses the stable storage when executing commit. The performance of the non-uniform algorithm can even compete with the crash-stop atomic broadcast algorithm (which does not access stable storage at all). As the commit period reduces, the performance of all crash-recovery algorithms, including the non-uniform algorithm, degrades asymptotically, since access to stable storage becomes more and more frequent.

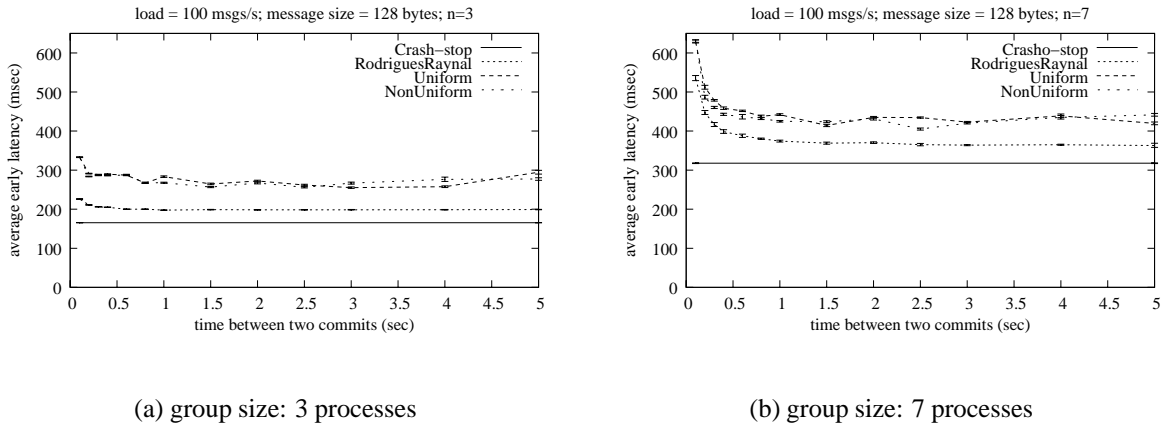


Figure 4: Early latency of various atomic broadcast algorithms

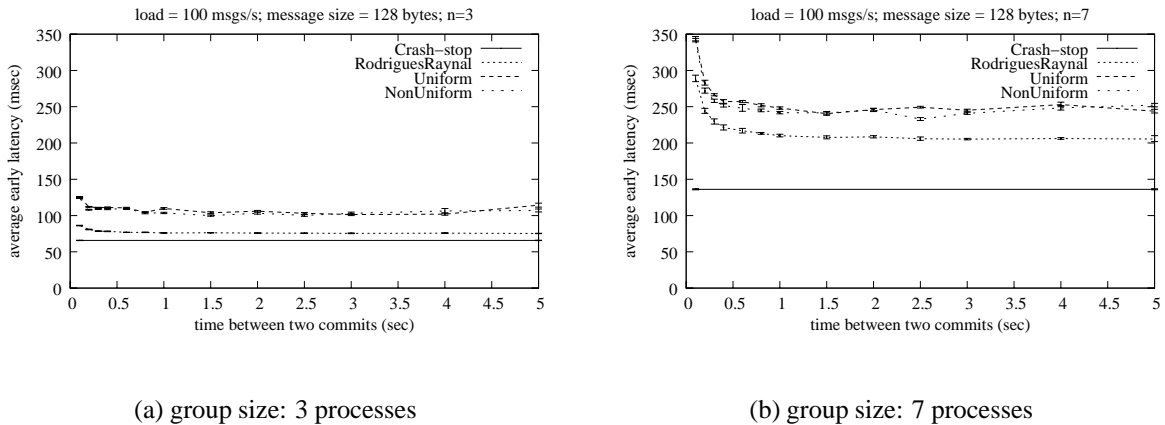


Figure 5:  $1 / \text{throughput}$  of various atomic broadcast algorithms

The throughput results are shown in Figure 5, also with the 95% confidence interval. Actually,

in order to compare the curves of Figures 4 and 5, we have plotted the values of  $1 / \text{throughput}$  in Figure 5. We can observe that the results in the two figures are very similar.

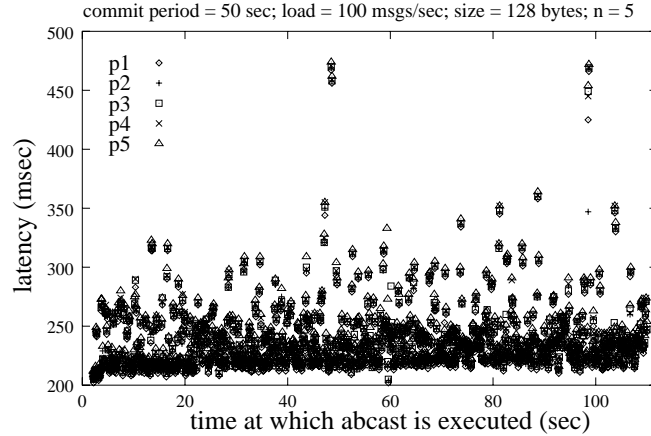


Figure 6: Latency in a single experiment with the non-uniform atomic broadcast algorithm.

Figures 4 and 5 do not allow us to directly spot the impact of *commit* on the latency. This can be seen in Figure 6. The figure corresponds to one single experiment with the non-uniform atomic broadcast for a group of size  $n = 5$ . The figure shows the latency, once the steady state is reached, as a function of the time at which the abcast is issued. The figure shows *all* latencies, not only the early latency: if the  $\text{abcast}(m)$  is issued at time  $t$  and  $m$  is delivered at  $p_1$  at time  $t + \Delta_1$ , at  $p_2$  at time  $t + \Delta_2$  at  $p_2$ , etc., we plot five dots with coordinates  $(t, \Delta_1), (t, \Delta_2), \dots, (t, \Delta_5)$ . In Figure 6, *commit* was executed approximately at  $t = 50$  and  $t = 100$ . We can clearly observe how latency is affected by the execution of *commit*. The high latencies around  $t = 50$  and  $t = 100$  come from messages that were already abcast but not yet delivered when the commit operation started: the latency of these messages was affected by the commit operation.

## 6 Conclusion

We have proposed two novel specifications of atomic broadcast in the crash-recovery model, for uniform and non-uniform atomic broadcast. The key point in these two specifications is the distinction between permanent and volatile events. This distinction allows us to properly define the concept of non-uniformity in the crash-recovery model. Despite some attempts in the literature [1, 4], the concept of non-uniformity in the crash-recovery model did not have so far a satisfactory definition. We have also pointed out the problem of process recovery after a crash, where the application state needs to

be consistent with the state of the atomic broadcast algorithm. We have shown how this problem can be solved. It is important to understand that this consistency problem does not arise in the crash-stop model, which explains that it was overlooked up to now. Finally, we have run experiments to compare the performance of the two new atomic broadcast algorithms with two published algorithms, one based on the crash-stop model, the other based on the crash-recovery model.

The specifications and algorithms given here are for static groups, i.e., for groups without membership change. In the future we plan to extend this work to dynamic groups, where processes can be added to and removed from the group during the computation. Dynamic groups have been considered in the crash-stop model, but not in the crash-recovery model.

## References

- [1] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.
- [2] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. On the performance of wide area synchronous database replication. Technical report, CNDS-2003-3. John Hopkins Univ., December 2003.
- [3] N. T. Bhatti, M. A. Hiltunen, and D. Schlichting. Coyote: A system for constructing fine-grain configurable communication services. *ACM Trans. on Computer Systems*, 16(4):321–366, November 1998.
- [4] R. Boichat and R. Guerraoui. Reliable broadcast in the crash-recovery model. In *Proc. of 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, Nuremberg, Germany, oct 2000.
- [5] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225–267, March 1996.
- [6] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2005.
- [7] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

- [8] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. TR 94-1425, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, May 1994.
- [9] M.A. Hiltunen and R.D. Schlichting. A configurable membership service. *IEEE Transactions on Computers*, 47(5):573–586, 1998.
- [10] M. Hurfin, A. Mostéfaoui, and M. Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proceedings of the 17th Symposium on Reliable Distributed Systems (SRDS)*, pages 280–286, West Lafayette, IN, USA, October 1998.
- [11] B. Kemme. *Database Replication for Clusters of Workstations*. PhD thesis, Swiss Federal Institute of Technology Zürich, Switzerland, August 2000. No. 13864.
- [12] S. Mena, X. Cuvellier, C. Grégoire, and A. Schiper. Appia vs. Cactus, comparing protocol composition frameworks. In *Proc. of 22th IEEE Symposium on Reliable Distributed Systems (SRDS'03)*, Florence, Italy, October 2003.
- [13] R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash-recover model. Technical Report 97/239, École Polytechnique Fédérale de Lausanne, Switzerland, August 1997.
- [14] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication. *IEEE Transactions on Knowledge and Data Engineering*, 15(4), 2003.
- [15] Péter Urbán. *Evaluating the Performance of Distributed Agreement Algorithms: Tools, Methodology and Case Studies*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, August 2003. Number 2824.
- [16] M. Wiesmann. *Group Communications and Database Replication: Techniques, Issues and Performance*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, May 2002. Number 2577.
- [17] M. Wiesmann and A. Schiper. Beyond 1-safety and 2-safety for replicated databases: Group-safety. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT2004)*, Heraklion - Crete - Greece, March 2004.